CVC3

The CVC3 User's Manual

Contents

What is CVC3?

CVC3 is an automated *validity checker* for a many-sorted (i.e., typed) first-order logic with *built-in theories*, including some support for quantifiers, partial functions, and predicate subtypes. The current built-in theories are the theories of:

- equality over *free* (aka *uninterpreted*) function and predicate symbols,
- real and integer linear arithmetic (with some support for non-linear arithmetic),
- bit vectors,
- arrays,
- tuples,
- records,
- user-defined inductive datatypes.

CVC3 checks whether a given formula ϕ is valid in the built-in theories under a given set Γ of assumptions. More precisely, it checks whether

$$\Gamma \models_T \phi$$

that is, whether ϕ is a logical consequence of the union T of the built-in theories and the set of formulas Γ.

Roughly speaking, when ϕ is *universal* and all the formulas in Γ are *existential* (i.e., when ϕ and Γ contain at most universal, respectively existential, quantifiers), **CVC3** is in fact a decision procedure: it is always guaranteed (well, modulo bugs and memory limits) to return a correct "valid" or "invalid" answer. In all other cases, **CVC3** is sound but incomplete: it will never say that an invalid formula is valid but it may either never return or give up and return "unknown" in some cases when $\Gamma \models_T \phi$.

When **CVC3** returns "valid" it can return a formal proof of the validity of ϕ under the *logical context* Γ, together with the subset Γ' of Γ used in the proof, such that $\Gamma' \models_T \phi$.

When **CVC3** returns "invalid" it can return, in the current terminology, both a *counter-example* to ϕ's validity under Γ and a *counter-model*. Both a counter-example and a counter-models are a set Δ of additional formulas consistent with Γ in T, but entailing the negation of ϕ. In formulas:

$$\Gamma \cup \Delta \not\models_T \mathit{false} \text{ and } \Gamma \cup \Delta \models_T \neg\phi.$$

The difference is that a counter-model is given as a set of equations providing a concrete assignment of values for the free symbols in Γ and ϕ (see **QUERY** for more details).

CVC3 can be used in two modes: as a library or as a command-line executable (implemented as a command-line interface to the library). Interfaces to the library are available in C/C++, Java and .NET. This manual mainly describes the command-line interface on a unix-type platform.

Running CVC3 from a Command Line

Assuming you have properly installed **CVC3** on your machine (check the **INSTALL** section for that), you will have an executable file called `cvc3`. It reads the input (a sequence of commands) from the standard input and prints the results on the standard output. Errors and some other messages (e.g. debugging traces) are printed on the standard error.

Typically, the input to `cvc3` is saved in a file and redirected to the executable, or given on a command line:

```
# Reading from standard input:
  cvc3 < input-file.cvc
# Reading directly from file:
  cvc3 input-file.cvc
```

Notice that, for efficiency, **CVC3** uses input buffers, and the input is not always processed immediately after each command. Therefore, if you want to type the commands interactively and receive immediate feedback, use the `+interactive` option (can be shortened to `+int`):

```
  cvc3 +int
```

Run `cvc3 -h` for more information on the available options.

The command line front-end of **CVC3** supports two input languages.

- CVC3's own *presentation language* whose syntax was initially inspired by the PVS and SAL systems and is almost identical to the input language of CVC and CVC Lite, the predecessors of **CVC3**;
- the standard language promoted by the SMT-LIB initiative for SMT-LIB benchmarks.

We describe the input languages next, concentrating mostly on the first.

Presentation Input Language

The input language consists of a sequence of symbol declarations and commands, each followed by a semicolon (`;`).

Any text after the first occurrence of a percent character and to the end of the current line is a comment:

```
%%% This is a CVC3 comment
```

Type system

CVC3's type system includes a set of built-in types which can be expanded with additional user-defined types.

The type system consists of *value* types, *non-value* types and *subtypes* of value types, all of which are interpreted as sets. For convenience, we will sometimes identify below the interpretation of a type with the type itself.

Value types consist of *atomic* types and *structured* types. The atomic types are REAL, $\mathrm{BITVECTOR}(n)$ for all $n > 0$, as well as user-defined atomic types (also called *uninterpreted* types). The structured types are *array*, *tuple*, and *record* types, as well as ML-style user-defined (inductive) *datatypes*.

Non-value types consist of the type $\mathrm{BOOLEAN}$ and *function* types. Subtypes include the built-in subtype INT of REAL and are covered in the **Subtypes** section below.

REAL Type

The REAL type is interpreted as the set of rational numbers. The name REAL is justified by the fact that a **CVC3** formula is valid in the theory of rational numbers iff it is valid in the theory of real numbers.

Bit Vector Types

For every positive numeral *n*, the type $\mathrm{BITVECTOR}(n)$ is interpreted as the set of all bit vectors of size *n*.

User-defined Atomic Types

User-defined atomic types are each interpreted as a set of unspecified cardinality but disjoint from any other type. They are created by declarations like the following:

```
% User declarations of atomic types:

MyBrandNewType: TYPE;

Apples, Oranges: TYPE;
```

BOOLEAN Type

The $\mathrm{BOOLEAN}$ type is, perhaps confusingly, the type of **CVC3** formulas, not the two-element set of Boolean values. The fact that $\mathrm{BOOLEAN}$ is not a value type in practice means that it is not possible for function symbols in **CVC3** to have a arguments of type $\mathrm{BOOLEAN}$. The reason is that **CVC3** follows the two-tiered structure of classical first-order logic that distinguishes between formulas and terms, and allows terms to occur in formulas but not vice versa. (An exception is the IF-THEN-ELSE construct, see later.) The only difference is that, syntactically, formulas in **CVC3** are terms of type $\mathrm{BOOLEAN}$. A function symbol f then *can* have $\mathrm{BOOLEAN}$ as its return type. But that is just CVC3's way, inherited from the previous systems of the CVC family, to say that f is a predicate symbol.

CVC3 does have a type that behaves like a Boolean Value type, that is, a value type with only two elements and with the usual Boolean operations defined on it: it is `BITVECTOR(1)`.

Function Types

All structured types are actually *families* of types. Function ($\rightarrow$) types are created by the mixfix type constructors

$$
\begin{aligned}
&\_ \rightarrow \_ \\
&(\_, \_) \rightarrow \_ \\
&(\_, \_, \_) \rightarrow \_ \\
&\ldots
\end{aligned}
$$

whose arguments can be instantiated by any value (sub)type, with the addition that the last argument can also

be $\mathrm{BOOLEAN}$.

```
% Function type declarations

UnaryFunType: TYPE = INT -> REAL;
BinaryFunType: TYPE = (REAL, REAL) -> ARRAY REAL OF REAL;
TernaryFunType: TYPE = (REAL, BITVECTOR(4), INT) -> BOOLEAN;
```

A function type of the form $(T_1, \ldots, T_n) \to T$ with $n > 0$ is interpreted as the set of all total functions from the Cartesian product $T_1 \times \cdots \times T_n$ to T when T is not $\mathrm{BOOLEAN}$. Otherwise, it is interpreted as the set of all relations over $T_1 \times \cdots \times T_n$

The example above also shows how to introduce *type names*. A name like UnaryFunType above is just an abbreviation for the type $\mathrm{INT} \to \mathrm{REAL}$ and can be used interchangeably with it.

In general, any type defined by a type expression E can be given a name with the declaration:

```
name : TYPE = E;
```

Array Types

Array types are created by the mixfix type constructors $\mathrm{ARRAY}\ \_\ \mathrm{OF}\ \_$ whose arguments can be instantiated by any value type.

```
T1 : TYPE;

% Array types:

ArrayType1: TYPE = ARRAY T1 OF REAL;
ArrayType2: TYPE = ARRAY INT OF (ARRAY INT OF REAL);
ArrayType3: TYPE = ARRAY [INT, INT] OF INT;
```

An array type of the form $\mathrm{ARRAY}\ T_1\ \mathrm{OF}\ T_2$ is interpreted as the set of all total maps from T_1 to T_2. The main conceptual difference with the type $T_1 \to T_2$ is that arrays, contrary to functions, are first-class objects of the language: they can be arguments or results of functions. Moreover, array types come equipped with an update operation.

Tuple Types

Tuple types are created by the mixfix type constructors

$$[\ \_\ ]$$
$$[\ \_\ ,\ \_\ ]$$
$$[\ \_\ ,\ \_\ ,\ \_\ ]$$
$$\ldots$$

whose arguments can be instantiated by any value type.

```
% Tuple declaration

TupleType: TYPE = [ REAL, ArrayType1, [INT, INT] ];
```

A tuple type of the form $[T_1, \ldots, T_n]$ is interpreted as the Cartesian product $T_1 \times \cdots \times T_n$. Note that while the types $(T_1, \ldots, T_n) \to T$ and $[T_1 \times \cdots \times T_n] \to T$ are semantically equivalent, they are operationally different in **CVC3**. The first is the type of functions that take *n* arguments, while the second is the type of functions of *1* argument of type n-tuple.

Record Types

Similar to, but more general than tuple types, record types are created by type constructors of the form

$$[\# \, l_1 : \_ \, , \, \ldots \, , \, l_n : \_ \, \#]$$

where $n > 0$, $l_1, \ldots, l_n$ are field labels, and the arguments can be instantiated with any value types.

```
% Record declaration

RecordType: TYPE = [# number: INT, value: REAL, info: TupleType #];
```

The order of the fields in a record type is meaningful. In other words, permuting the field names gives a different type. Note that records are

non-recursive. For instance, it is not possible to declare a record type called `Person` containing a field of type `Person`. Recursive types are provided in **CVC3** as ML-style datatypes.

Inductive Data Types

Inductive datatypes are created by declarations of the form

$$\begin{array}{l} \text{DATATYPE} \\ \quad type\_name_1 = C_{1,1} \mid C_{1,2} \mid \cdots \mid C_{1,m_1}, \\ \quad type\_name_2 = C_{2,1} \mid C_{2,2} \mid \cdots \mid C_{2,m_2}, \\ \quad \vdots \\ \quad type\_name_n = C_{n,1} \mid C_{n,2} \mid \cdots \mid C_{n,m_n} \\ \text{END;} \end{array}$$

Each of the C_{ij} is either a constant symbol or an expression of the form

$$cons(\ sel_1 : T_1, \ \ldots, \ sel_k : T_k\ )$$

where $T_1, \ldots, T_k$ are any value types or type names for value types, including any $type\_name_i$. Such declarations introduce for the datatype:

- *constructor* symbols $cons$ of type $(T_1, \ldots, T_k) \to type\_name_i$,
- *selector* symbols sel_j of type $type\_name_i \to T_j$, and
- *tester* symbols is_cons of type $type\_name_i \to \text{BOOLEAN}$.

Here are some examples of datatype declarations:

```
% simple enumeration type
% implicitly defined are the testers: is_red, is_yellow and is_blue
% (similarly for the other datatypes)

DATATYPE
  PrimaryColor = red | yellow | blue
END;


% infinite set of pairwise distinct values ...v(-1), v(0), v(1), ...

DATATYPE
  Id = v (id: INT)
END;


% ML-style integer lists
```

```
DATATYPE
  IntList = nil | cons (head: INT, tail: IntList)
END;


% ASTs

DATATYPE
  Term = var (index: INT)
       | apply (arg_1: Term, arg_2: Term)
       | lambda (arg: INT, body: Term)
END;


% Trees

DATATYPE
  Tree = tree (value: REAL, children: TreeList),
  TreeList = nil_tl
           | cons_tl (first_t1: Tree, rest_t1: TreeList)
END;
```

Constructor, selector and tester symbols defined for a datatype have global scope. So, for instance, it is not possible for two different datatypes to use the same name for a constructor.

A datatype is interpreted as a term algebra constructed by the constructor symbols over some sets of generators. For example, the datatype `IntList` is interpreted as the set of all terms constructed with `nil` and `cons` over the integers.

Because of this semantics, **CVC3** allows only *inductive* datatypes, that is, datatypes whose values are essentially (labeled, ordered) finite trees. Infinite structures such as streams or even finite but cyclic ones such as circular lists are then excluded. For instance, none of the following declarations define inductive datatypes, and are rejected by **CVC3**:

```
DATATYPE
 IntStream = s (first:INT, rest: IntStream)
END;

DATATYPE
 RationalTree = node1 (first_child1: RationalTree)
              | node2 (first_child2: RationalTree, second_child2:RationalTree)
END;

DATATYPE
 T1 =  c1 (s1: T2),
 T2 =  c2 (s2: T1)
END;
```

In concrete, a declaration of $n \geq 1$ datatypes $T_1, \ldots, T_n$ will be rejected if for any one of the types $T_1, \ldots, T_n$, it is impossible to build a finite term of that type using only the constructors of $T_1, \ldots, T_n$ and free constants of type other than $T_1, \ldots, T_n$.

Datatypes are the only types for which the user also chooses names for the built-in operations defined on the type for:

- constructing a value (with the constructors),
- extracting components from a value (with the selectors), or
- checking if a value was constructed with a certain constructor or not (with the testers).

For all the other types, **CVC3** provides predefined names for the built-in operations on the type.

Type Checking

In essence, **CVC3** terms are statically typed at the level of types--as opposed to *sub*types--according to the usual rules of first-order many-sorted logic (the typing rules for formulas are analogous):

- each variable has one associated (non-function) type,
- each constant symbol has one associated (non-function) type,
- each function symbol has one or more associated function types,
- the type of a term consisting just of a variable or a constant symbol is the type associated to that variable or constant symbol,
- the term obtained by applying a function symbol f to the terms $t_1, \ldots, t_n$ is T if f has type $(T_1, \ldots, T_n) \to T$ and each t_i has type T_i.

Attempting to enter an ill-typed term will result in an error.

The main difference with standard many-sorted logic is that some built-in symbols are parametrically polymorphic. For instance the function symbol for extracting the element of any array has type $(ARRAY\ T_1\ OF\ T_2,\ T_1) \to T_2$ for all types T_1, T_2 not containing function or predicate types.

Terms and Formulas

In addition to type expressions, **CVC3** has expressions for terms and formulas (i.e., terms of type $\mathrm{BOOLEAN}$). By and large, these are standard first-order terms built out of (typed) variables, predefined theory-specific operators, free (i.e., user-defined) function symbols, and quantifiers. Extensions include an if-then-else operator, lambda abstractions, and local symbol declarations, as illustrated below. Note that these extensions still keep CVC3's language first-order. In particular, lambda abstractions are restricted to take and return only terms of a value type. Similarly, quantifiers can only quantify variables of a value type.

Free function symbols include *constant* symbols and *predicate* symbols, respectively nullary function symbols and function symbols with a $\mathrm{BOOLEAN}$ return type. Free symbols are introduced with global declarations of the form $f_1, \ldots, f_m : T$; where $m > 0$, f_i are the names of the symbols and T is their type:

```
% integer constants

a, b, c: INT;

% real constants

x,y,z: REAL;

% unary function

f1: REAL -> REAL;

% binary function

f2: (REAL, INT) -> REAL;

% unary function with a tuple argument

f3: [INT, REAL] -> BOOLEAN;

% binary predicate

p: (INT, REAL) -> BOOLEAN;

% Propositional "variables"

P,Q; BOOLEAN;
```

Like type declarations, such free symbol declarations have global scope and must be unique. In other words, it is

not possible to globally declare a symbol more than once. This entails among other things that free symbols cannot be overloaded with different types.

As with types, a new free symbol can be defined as the name of a term of the corresponding type. With constant symbols this is done with a declaration of the form $f : T = t;$:

```
c: INT;

i: INT = 5 + 3*c;

j: REAL = 3/4;

t: [REAL, INT] = (2/3, -4);

r: [# key: INT, value: REAL #] = (# key := 4, value := (c + 1)/2 #);

f: BOOLEAN = FORALL (x:INT): x <= 0 OR x > c ;
```

A restriction on constants of type $BOOLEAN$ is that their value can only be a *closed* formula, that is, a formula with no free variables.

A term and its name can be used interchangeably in later expressions. Named terms are often useful for shared subterms (terms used several times in different places) since their use can make the input exponentially more concise. Named terms are processed very efficiently by **CVC3**. It is much more efficient to associate a complex term with a name directly rather than to declare a constant and later assert that it is equal to the same term. This point will be explained in more detail later in section **Commands**.

More generally, in **CVC3** one can associate a term to function symbols of any arity. For non-constant function symbols this is done with a declaration of the form

$$f : (T_1, \ldots, T_n) \to T = \text{LAMBDA}(x_1 : T_1, \ldots, x : T_n) : t\,;$$

where t is any term of type T with free variables in $\{x_1, \ldots, x_n\}$. The lambda binder has the usual semantics and conforms to the usual lexical scoping rules: within the term t the declaration of the symbols $x_1, \ldots, x_n$ as local variables of respective type $T_1, \ldots, T_n$ hides any previous, global declaration of those symbols.

As a general shorthand, when k consecutive types $T_i, \ldots, T_{i+k-1}$ in the lambda expression $\text{LAMBDA}(x_1 : T_1, \ldots, x : T_n) : t$ are identical, the syntax $\text{LAMBDA}(x_1 : T_1, \ldots, x_i, \ldots, x_{i+k-1} : T_i, \ldots, x : T_n) : t$ is also allowed.

```
% Global declaration of x as a unary function symbol

x: REAL -> REAL;


% Local declarations of x as a constant symbol

f: REAL -> REAL = LAMBDA (x: REAL): 2*x + 3;

p: (INT, INT) -> BOOLEAN = LAMBDA (x,i: INT): i*x - 1 > 0;

g: (REAL, INT) -> [REAL, INT] = LAMBDA (x: REAL, i:INT): (x + 1, i - 3);
```

Constant and function symbols can also be declared *locally* anywhere within a term by means of a *let* binder. This is done with a declaration of the form

$$\begin{array}{ll} \text{LET} & f_1 = t_1, \\ & \vdots \\ & f_n = t_m \\ \text{IN} & t; \end{array}$$

for constant symbols, and of the form

$$\begin{array}{rcl} \text{LET} \quad f_1 & = & \text{LAMBDA}(x_1^1 : T_1^1, \ldots, x_1^{n_1} : T_1^{n_1}) : t_1, \\ & \vdots & \\ f_m & = & \text{LAMBDA}(x_m^1 : T_m^1, \ldots, x_m^{n_m} : T_m^{n_m}) : t_m \\ \text{IN} \quad t; & & \end{array}$$

for non-constant symbols. Let binders can be nested arbitrarily and follow the usual lexical scoping rules.

```
t: REAL =
  LET g = LAMBDA(x:INT): x + 1,
      x1 = 42,
      x2 = 2*x1 + 7/2
  IN
     (LET x3 = g(x1) IN x3 + x2) / x1;
```

Note that the same symbol = is used, unambiguously, in the syntax of global declarations, let declarations, and as a predicate symbol.

In addition to user-defined symbols, **CVC3** terms can use a number of predefined symbols: the logical symbols as well as *theory* symbols, function symbols belonging to one of the built-in theories. They are described next, with the theory symbols grouped by theory.

Logical Symbols

The logical symbols in CVC3's language include the equality and disequality predicate symbols, respectively written as = and /=, the multiarity disequality symbol DISTINCT, together with the logical constants TRUE, FALSE, the connectives NOT, AND, OR, XOR, =>, <=>, and the first-order quantifiers EXISTS and FORALL, all with the standard many-sorted logic semantics.

The binary connectives have infix syntax and type $(\text{BOOLEAN}, \text{BOOLEAN}) \to \text{BOOLEAN}$. The symbols = and /=, which are also infix, are instead polymorphic, having type $(T, T) \to \text{BOOLEAN}$ for every predefined or user-defined value type T. They are interpreted respectively as the identity relation and its complement.

The DISTINCT symbol is both overloaded and polymorphic. It has type $(T, ..., T) \to \text{BOOLEAN}$ for every tuple $(T, ..., T)$ of length $n > 0$ where T is a predefined or user-defined value type. For each $n > 0$, it is interpreteted as the relation that holds exactly for tuples of pairwise distinct elements.

The syntax for quantifiers is similar to that of the lambda binder.

Here is an example of a formula built just of these logical symbols and variables:

```
A, B: TYPE;

quant: BOOLEAN = FORALL (x,y: A, i,j,k: B): i = j AND i /= k
                   => EXISTS (z: A): x /= z OR z /= y;
```

Binding and scoping of quantified variables follows the same rules as in let expressions. In particular, a quantifier will shadow in its scope any constant and function symbols with the same name as one of the variables it quantifies:

```
A: TYPE;
i,j: INT;

% The first occurrence of i and of j in f are constant symbols,
% the others are variables.

f: BOOLEAN =  i = j AND FORALL (i,j: A): i = j OR i /= j;
```

Optionally, it is also possible to specify *instantiation patterns* for quantified variables. The general syntax for a

quantified formula ψ with patterns is

$$Q\,(x_1 : T_1, \ldots, x_k : T_k) :\ p_1 : \ldots\ p_n :\ \varphi$$

where $n \geq 0$, Q is either FORALL or EXISTS, φ is a term of type $\mathrm{BOOLEAN}$, and each of the p_i's, a *pattern for the quantifier* $Q\,(x_1 : T_1, \ldots, x_k : T_k)$, has the form

$$\mathrm{PATTERN}\ (t_1, \ldots, t_m)$$

where $m > 0$ and $t_1, \ldots, t_m$ are arbitrary binder-free terms (no lets, no quantifiers). Those terms can contain (free) variables, typically, but not exclusively, drawn from $x_1, \ldots, x_k$. (Additional variables can occur if ψ occurs in a bigger formula binding those variables.)

```
A: TYPE;
b, c: A;
p, q: A -> BOOLEAN;
r: (A, A) -> BOOLEAN;

ASSERT FORALL (x0, x1, x2: A):
         PATTERN (r(x0, x1), r(x1, x2)):
         (r(x0, x1) AND r(x1, x2)) => r(x0, x2) ;

ASSERT FORALL (x: A):
         PATTERN (r(x, b)):
         PATTERN (r(x, c)):
         p(x) => q(x) ;

ASSERT EXISTS (y: A):
         FORALL (x: A):
           PATTERN (r(x, y), p(y)):
           r(x, y) => q(x) ;
```

Patterns have no logical meaning: adding them to a formula does not change its semantics. Their purpose is purely operational, as explained in Section **Instantiation Patterns**.

In addition to these constructs, **CVC3** also has a general mixfix conditional operator of the form

$$\mathrm{IF}\ b\ \mathrm{THEN}\ t\ \mathrm{ELSIF}\ b_1\ \mathrm{THEN}\ t_1\ \ldots\ \mathrm{ELSIF}\ b_n\ \mathrm{THEN}\ t_n\ \mathrm{ELSE}\ t_{n+1}\ \mathrm{ENDIF}$$

with $n \geq 0$ where $b, b_1, \ldots, b_n$ are terms of type $\mathrm{BOOLEAN}$ and $t, t_1, \ldots, t_n, t_{n+1}$ are terms of the same value type T:

```
% Conditional term
x,y,z,w:REAL;
t: REAL =
  IF x > 0 THEN y
  ELSIF x >= 1 THEN z
  ELSIF x > 2 THEN w
  ELSE 2/3 ENDIF;
```

User-defined Functions and Types

The theory of user-defined functions is in effect a family of theories of equality parametrized by the atomic types and the free symbols a user can define during a run of **CVC3**.

The theory's function symbols consist of all *and only* the user-defined free symbols.

Arithmetic

The real arithmetic theory has predefined symbols for the usual arithmetic constants and operators over the type REAL, each with the expected type: all numerals 0, 1, ..., as well as - (both unary and binary), +, *, /, <, >, <=, >=. Rational values can be expressed in fractional form: e.g., 1/2, 3/4, etc.

The size of numerals used in the representation of natural and rational numbers is unbounded (or more accurately, bounded only by the amount of available memory).

Bit vectors

The bit vector theory has a large number of predefined function symbols denoting various bit vector operators. We describe the operators and their semantics informally below, often omitting a specification of their type, which should be easy to infer.

The operators' names are overloaded in the obvious way. For instance, the same name is used for each $m, n > 0$ for the operator that takes a bit vector of size m and one of size n and returns their concatenation.

For each size n, there are 2^n elements in the type $\mathrm{BITVECTOR}(n)$. These elements can be named using constant symbols or *bit vector constants*. Each element in the domain is named by two different constant symbols: once in binary and once in hexadecimal format. Binary constant symbols start with the characters 0bin and continue with the representation of the vector in the usual binary format (as an n-string over the characters 0,1). Hexadecimal constant symbols start with the characters 0hex and continue with the representation of the vector in usual hexadecimal format (as an n-string over the characters 0,...,9,a,...,f).

```
Binary constant          Corresponding hexadecimal constant
-----------------------------------------------------------
0bin0000111101010000     0hex0f50
```

In the binary representation, the rightmost bit is the least significant bit (LSB) of the vector and the leftmost bit is the most significant bit (MSB). The index of the LSB in the bit vector is 0 and the index of the MSB is n-1 for an n-bit constant. This convention extends to all bit vector expressions in the natural way.

Bit vector operators are categorized into word-level, bitwise, arithmetic, and comparison operators.

```
WORD-LEVEL OPERATORS:

Description      Symbol            Example
=================================================================
Concatenation   _ @ _             0bin01@0bin0          (= 0bin010)
Extraction      _ [i:j]           0bin0011[3:1]         (= 0bin001)
Left shift      _ << k            0bin0011 << 3         (= 0bin0011000)
Right shift     _ >> k            0bin1000 >> 3         (= 0bin0001)
Sign extension  SX(_,k)           SX(0bin100, 5)        (= 0bin11100)
Zero extension  BVZEROEXTEND(_,k) BVZEROEXTEND(0bin1,3) (= 0bin0001)
Repeat          BVREPEAT(_,k)     BVREPEAT(0bin10,3)    (= 0bin101010)
Rotate left     BVROTL(_,k)       BVROTL(0bin101,1)     (= 0bin011)
Rotate right    BVROTR(_,k)       BVROTR(0bin101,1)     (= 0bin110)
```

For each $m, n > 0$ there is

- one infix concatenation operator, taking an m-bit vector v_1 and an n-bit vector v_2 and returning the $(m+n)$-bit concatenation of v_1 and v_2;
- one postfix extraction operator $[i:j]$ for each i, j with $n > i >= j >= 0$, taking an n-bit vector v and returning the $(i-j+1)$-bit subvector of v at positions i through j (inclusive);
- one postfix left shift operator $<< k$ for each $k >= 0$, taking an n-bit vector v and returning the $(n+k)$-bit concatenation of v with the k-bit zero vector;
- one postfix right shift operator $>> k$ for each $k >= 0$, taking an n-bit vector v and returning the n-bit concatenation of the k-bit zero bit vector with $v[n-1:k]$;
- one mixfix sign extension operator $\mathrm{SX}(\_, k)$ for each $k >= n$, taking an n-bit vector v and returning the k-bit concatenation of $k-n$ copies of the MSB of v and v.
- one mixfix zero extension operator $\mathrm{BVZEROEXTEND}(\_, k)$ for each $k >= 1$, taking an n-bit vector v

and returning the $n+k$-bit concatenation of k zeroes and v.
- one mixfix repeat operator $\mathrm{BVREPEAT}(\_,k)$ for each $k >= 1$, taking an n-bit vector v and returning the $n*k$-bit concatenation of k copies of v.
- one mixfix rotate left operator $\mathrm{BVROTL}(\_,k)$ for each $k >= 0$, taking an n-bit vector v and returning the (n)-bit vector obtained by rotating the bits of v left k times, where a single rotation means removing the MSB and concatenating it as the new LSB.
- one mixfix rotate right operator $\mathrm{BVROTR}(\_,k)$ for each $k >= 0$, taking an n-bit vector v and returning the (n)-bit vector obtained by rotating the bits of v right k times, where a single rotation means removing the LSB and concatenating it as the new MSB.

```
BITWISE OPERATORS:

Description       Symbol
============================
Bitwise AND       _ & _
Bitwise OR        _ | _
Bitwise NOT       ~ _
Bitwise XOR       BVXOR(_,_)
Bitwise NAND      BVNAND(_,_)
Bitwise NOR       BVNOR(_,_)
Bitwise XNOR      BVXNOR(_,_)
Bitwise Compare   BVCOMP(_,_)
```

For each $n > 0$ there are operators with the names and syntax above, performing the usual bitwise Boolean operations on n-bit arguments. All produce n-bit results except for BVCOMP which always produces a 1-bit result: $0\mathrm{bin}1$ if its two arguments are equal and $0\mathrm{bin}0$ otherwise.

```
ARITHMETIC OPERATORS:

Description               Symbol
=======================================
Bit vector addition            BVPLUS(k,_,_,...)
Bit vector multiplication      BVMULT(k,_,_)
Bit vector negation            BVUMINUS(_)
Bit vector subtraction         BVSUB(k,_,_)
Bit vector left shift          BVSHL(_,_)
Bit vector arith shift right   BVASHR(_,_)
Bit vector logic shift right   BVLSHR(_,_)
Bit vector unsigned divide     BVUDIV(_,_)
Bit vector signed divide       BVSDIV(_,_)
Bit vector unsigned remainder  BVUREM(_,_)
Bit vector signed remainder    BVSREM(_,_)
Bit vector signed modulus      BVSMOD(_,_)
```

For each $n > 0$ and $k > 0$ there is

- one addition operator $\mathrm{BVPLUS}(k,\_,\_,\ldots)$, taking two or more bit vectors of arbitrary size, and returning the (k) least significant bits of their sum.
- one multiplication operator $\mathrm{BVMULT}(k,\_,\_)$, taking two bit vectors v_1 and v_2, and returning the k least significant bits of their product.
- one prefix negation operator $\mathrm{BVUMINUS}(\_)$, taking an n-bit vector v and returning the n-bit vector $\mathrm{BVPLUS}(n,\tilde{}v,0\mathrm{bin}1)$.
- one subtraction operator $\mathrm{BVSUB}(k,\_,\_)$, taking two bit vectors v_1 and v_2, and returning the k-bit vector $\mathrm{BVPLUS}(k,v_1,\mathrm{BVUMINUS}(v'))$ where v' is v_2 if the size of v_2 is greater than or equal to k, and v_2 extended to size k by concatenating zeroes in the most significant bits otherwise.
- one left shift operator $\mathrm{BVSHL}(\_,\_)$, taking two n-bit vectors v_1 and v_2, and returning the n-bit vector obtained by creating a vector of zeroes whose length is the value of v_2, concatenating this vector onto the least significant bits of v_1, and then taking the least significant n bits of the result.

- one arithmetic shift right operator $\mathrm{BVASHR}(\_,\_)$, taking two n-bit vectors v_1 and v_2, and returning the n-bit vector obtained by creating a vector whose length is the value of v_2 and each of whose bits has the same value as the MSB of v_1, concatenating this vector onto the most significant bits of v_1, and then taking the most significant n bits of the result.
- one logical shift right operator $\mathrm{BVLSHR}(\_,\_)$, taking two n-bit vectors v_1 and v_2, and returning the n-bit vector obtained by creating a vector of zeroes whose length is the value of v_2, concatenating this vector onto the most significant bits of v_1, and then taking the most significant n bits of the result.
- one unsigned integer division operator $\mathrm{BVUDIV}(\_,\_)$, taking two n-bit vectors v_1 and v_2, and returning the n-bit vector that is the largest integer value that can be multiplied by the integer value of v_2 to obtain an integer less than or equal to the integer value of v_1.
- one signed integer division operator $\mathrm{BVSDIV}(\_,\_)$.
- one unsigned integer remainder operator $\mathrm{BVUREM}(\_,\_)$.
- one signed integer remainder operator $\mathrm{BVSREM}(\_,\_)$ (sign follows dividend).
- one signed integer modulus operator $\mathrm{BVSMOD}(\_,\_)$ (sign follows divisor).

For precise definitions of the last four operators, we refer the reader to the equivalent operators defined in the SMT-LIB QF_BV logic (**SMT-LIB Input Language**).

CVC3 does not have dedicated operators for multiplexers. However, specific multiplexers can be easily defined with the aid of conditional terms.

```
% Example of 2-to-1 multiplexer

mp: (BITVECTOR(1), BITVECTOR(1), BITVECTOR(1)) -> BITVECTOR(1) =
      LAMBDA (s,x,y : BITVECTOR(1)): IF s = 0bin0 THEN x ELSE y ENDIF;
```

In addition to equality and disequality, **CVC3** provides the following comparison operators.

```
COMPARISON OPERATORS:

Description              Symbol
==================================
Less than                      BVLT(_,_)
Less than or equal to          BVLE(_,_)
Greater than                   BVGT(_,_)
Greater than equal to          BVGE(_,_)
Signed less than               BVSLT(_,_)
Signed less than or equal to   BVSLE(_,_)
Signed greater than            BVSGT(_,_)
Signed greater than equal to   BVSGE(_,_)
```

For each $m, n > 0$ there is

- one prefix "less than" operator $\mathrm{BVLT}(\_,\_)$, taking an m-bit vector v_1 and an n-bit vector v_2, and having the value TRUE iff the zero-extension of v_1 to k bits is less than the zero-extension of v_2 to k bits, where k is the maximum of m and n.
- one prefix "less than or equal to" operator $\mathrm{BVLE}(\_,\_)$, taking an m-bit vector v_1 and an n-bit vector v_2, and having the value TRUE iff the zero-extension of v_1 to k bits is less than or equal to the zero-extension of v_2 to k bits, where k is the maximum of m and n.
- one prefix "greater than" operator $\mathrm{BVGT}(\_,\_)$, taking an m-bit vector v_1 and an n-bit vector v_2, and having the same value as $\mathrm{BVLT}(v_2, v_1)$.
- one prefix "greater than or equal to" operator $\mathrm{BVGE}(\_,\_)$, taking an m-bit vector v_1 and an n-bit vector v_2, and having the same value as $\mathrm{BVLE}(v_2, v_1)$.

The signed operators are similar except that the values being compared are considered to be signed bit vector representations (in 2's complement) of integers.

Following are some example **CVC3** input formulas involving bit vector expressions

Example 1 illustrates the use of arithmetic, word-level and bitwise NOT operations:

```
x : BITVECTOR(5);
y : BITVECTOR(4);
yy : BITVECTOR(3);

QUERY
 BVPLUS(9, x@0bin0000, (0bin000@(~y)@0bin11))[8:4] = BVPLUS(5, x, ~(y[3:2])) ;
```

Example 2 illustrates the use of arithmetic, word-level and multiplexer terms:

```
bv : BITVECTOR(10);
a : BOOLEAN;

QUERY
 0bin01100000[5:3]=(0bin1111001@bv[0:0])[4:2]
 AND
 0bin1@(IF a THEN 0bin0 ELSE 0bin1 ENDIF) = (IF a THEN 0bin110 ELSE 0bin011 ENDIF)[1:0] ;
```

Example 3 illustrates the use of bitwise operations:

```
x, y, z, t, q : BITVECTOR(1024);

ASSERT x = ~x;
ASSERT x&y&t&z&q = x;
ASSERT x|y = t;
ASSERT BVXOR(x,~x) = t;
QUERY  FALSE;
```

Example 4 illustrates the use of predicates and all the arithmetic operations:

```
x, y : BITVECTOR(4);

ASSERT x = 0hex5;
ASSERT y = 0bin0101;

QUERY
 BVMULT(8,x,y)=BVMULT(8,y,x)
 AND
 NOT(BVLT(x,y))
 AND
 BVLE(BVSUB(8,x,y), BVPLUS(8, x, BVUMINUS(x)))
 AND
 x = BVSUB(4, BVUMINUS(x), BVPLUS(4, x,0hex1)) ;
```

Example 5 illustrates the use of shift functions

```
x, y : BITVECTOR(8);
z, t : BITVECTOR(12);

ASSERT x = 0hexff;
ASSERT z = 0hexff0;
QUERY  z = x << 4;
QUERY (z >> 4)[7:0] = x;
```

Arrays

The theory of arrays is a parametric theory of (total) unary functions. It comes equipped with polymorphic selection and update operators, respectively

$\_[\_]$ and $\_$ WITH $[\_] := \_$

with the usual semantics. For each *index* type T_1 and *element* type T_2, the first operator maps an array from T_1 to T_2 and an index into it (i.e., a value of type T_1) to the element of type T_2 "stored" into the array at that index. The second maps an array a from T_1 to T_2, an index i, and a T_2-element e to the array that stores e at index i and is otherwise identical to a.

Since arrays are just maps, equality between them is *extensional*: for two arrays of the same type to be different they have to store differ elements in at least one place.

Sequential updates can be chained with the syntax $\_$ WITH $[\_] := \_, \ldots, [\_] := \_$.

```
A: TYPE = ARRAY INT OF REAL;
a: A;
i: INT = 4;

% selection:


elem: REAL = a[i];

% update

a1: A = a WITH [10] := 1/2;

% sequential update
% (syntactic sugar for (a WITH [10] := 2/3) WITH [42] := 3/2)

a2: A = a WITH [10] := 2/3, [42] := 3/2;
```

Datatypes

The theory of datatypes is in fact a *family* of theories parametrized by a datatype declaration specifying constructors and selectors for a particular datatype.

No built-in operators other than equality and disequality are provided for this family in the presentation language. Each datatype declaration, however, generates constructor, selector and tester operators as described in Section **Inductive Data Types**.

Tuples and Records

Although they are currently implemented separately in **CVC3**, semantically both records and tuples can be seen as special instances of datatypes. In fact, a record of type $[\#l_0 : T_0, \ldots, l_n : T_n\#]$ could be equivalently modeled as, say, the datatype

$$\begin{array}{l}\textbf{DATATYPE}\\ \quad \text{Record} = rec(l_0 : T_0, \ldots, l_n : T_n)\\ \textbf{END;}\end{array}$$

Tuples could be seen in turn as special cases of records where the field names are the numbers from 0 to the length of the tuple minus 1. Currently, however, tuples and records have their own syntax for constructor and selector operators.

Records of type $[\#l_0 : T_0, \ldots, l_n : T_n\#]$ have the associated built-in constructor $(\#\ l_0 := \_, \ldots, l_n := \_\ \#)$ whose arguments must be terms of type $T_0, \ldots, T_n$, respectively.

Tuples of type $[\ T_0, \ldots, T_n\ ]$ have the associated built-in constructor $(\ \_, \ldots, \_\ )$ whose arguments must be terms of type $T_0, \ldots, T_n$, respectively.

The selector operators on records and tuples follows a dot notation syntax.

```
% Record construction and field selection

Item: TYPE = [# key: INT, weight: REAL #];

x: Item = (# key := 23, weight := 43/10 #);
k: INT = x.key;
v: REAL = x.weight;

% Tuple construction and projection

y: [REAL,INT,REAL] = ( 4/5, 9, 11/9 );
first_elem: REAL = y.0;
third_elem: REAL = y.2;
```

Differently from datatypes, records and tuples are also provided with built-in update operators similar in syntax and semantics to the update operator for arrays. More precisely, for each record type $[\# \, l_0 : T_0, \ldots, l_n : T_n \, \#]$ and each $i = 0, \ldots, n$, **CVC3** provides the operator

$$\_ \text{ WITH } .l_i \ := \ \_$$

The operator maps a record r of that type and a value v of type T_i to the record that stores v in field l_i and is otherwise identical to r. Analogously, for each tuple type $[T_0, \ldots, T_n]$ and each $i = 0, \ldots, n$, **CVC3** provides the operator

$$\_ \text{ WITH } .i \ := \ \_$$

```
% Record updates

Item: TYPE = [# key: INT, weight: REAL #];

x:  Item = (# key := 23, weight := 43/10 #);

x1: Item = x WITH .weight := 48;

% Tuple updates

Tup: TYPE = [REAL,INT,REAL];
y:  Tup = ( 4/5, 9, 11/9 );
y1: Tup = y WITH .1 := 3;
```

Updates to a nested component can be combined in a single WITH operator:

```
Cache: TYPE = ARRAY [0..100] OF [# addr: INT, data: REAL #];
State: TYPE = [# pc: INT, cache: Cache #];

s0: State;
s1: State = s0 WITH .cache[10].data := 2/3;
```

Note that, differently from updates on arrays, tuple and record updates are just additional syntactic sugar. For instance, the record x1 and tuple y1 defined above could have been equivalently defined as follows:

```
% Record updates

Item: TYPE = [# key: INT, weight: REAL #];

x:  Item = (# key := 23, weight := 43/10 #);

x1: Item = (# key := x.key,  weight := 48 #);
```

```
% Tuple updates

Tup: TYPE = [REAL,INT,REAL];
y:  Tup = ( 4/5, 9, 11/9 );
y1: Tup = ( y.0, 3, y.1 );
```

Commands

In addition to declarations of types and constants, the **CVC3** input language contains the following commands:

- ASSERT F -- Add the formula F to the current logical context Γ.
- QUERY F -- Check if the formula F is valid in the current logical context: $\Gamma \models_T F$.
- CHECKSAT F -- Check if the formula is satisfiable in the current logical context: $\Gamma \cup \{F\} \not\models_T false$.
- WHERE -- Print all the assumptions in the current logical context Γ.
- COUNTEREXAMPLE -- After an invalid QUERY or satisfiable CHECKSAT, print the context that is a witness for invalidity/satisfiability.
- COUNTERMODEL -- After an invalid QUERY or satisfiable CHECKSAT, print a model that makes the formula invalid/satisfiable. The model is in terms of concrete values for each free symbol.
- CONTINUE -- Search for a counter-example different from the current one (after an invalid QUERY or satisfiable CHECKSAT).
- RESTART F -- Restart an invalid QUERY or satisfiable CHECKSAT with the additional assumption F.

- PUSH -- Save (checkpoint) the current state of the system.
- POP -- Restore the system to the state it was in right before the last call to PUSH
- POPTO n -- Restore the system to the state it was in right before the most recent call to PUSH made from stack level n. Note that the current stack level is printed as part of the output of the WHERE command.

- TRANSFORM F -- Simplify F and print the result.
- PRINT F -- Parse and print back the expression F.
- OPTION *option value* -- Set the command-line option flag *option* to *value*. Note that *option* is given as a string enclosed in double-quotes and *value* as an integer.

The remaining commands take a single argument, given as a string enclosed in double-quotes.

- TRACE *flag* -- Turn on tracing for the debug flag *flag*.
- UNTRACE *flag* -- Turn off tracing for the debug flag *flag*.

- ECHO *string* -- Print *string*
- INCLUDE *filename* -- Read commands from the file *filename*.

Here, we explain some of the above commands in more detail.

QUERY

The command QUERY F invokes the core functionality of **CVC3** to check the validity of the formula F with respect to the assertions made thus far (Γ). F should be a formula as described in **Terms and Formulas**.

There are three possible answers.

- When the answer is "Valid", this means that $\Gamma \models_T F$. After a valid query, the logical context Γ is exactly as it was before the query.
- When the answer is "Invalid", this means that $\Gamma \not\models_T F$. In other words, there is a model of T satisfying $\Gamma \cup \{\neg F\}$. After an invalid query, the logical context Γ is augmented with new literals Δ such that $\Gamma \cup \Delta$ is consistent in the theory T, but $\Gamma \cup \Delta \models_T \neg F$. In fact, in this case $\Gamma \cup \Delta$ *propositionally* satisfies $\neg f$. We call the new context $\Gamma \cup \Delta$ a *counterexample* for F.
- An answer of "Unknown" is very similar to an answer of "Invalid" in that additional literals are added to the context which propositionally falsify the query formula F. The difference is that because **CVC3** is incomplete for some theories, it cannot guarantee in this case that $\Gamma \cup \Delta$ is actually consistent in T. The only sources of incompleteness in **CVC3** are non-linear arithmetic and quantifiers.

Counterexamples can be printed out using WHERE or COUNTEREXAMPLE commands. WHERE always prints out all of $\Gamma \cup C$. COUNTEREXAMPLE may sometimes be more selective, printing a subset of those formulas from the context

which are sufficient for a counterexample.

Since the QUERY command may modify the current context, if you need to check several formulas in a row in the same context, it is a good idea to surround every QUERY command by PUSH and POP in order to preserve the context:

```
PUSH;
QUERY <formula>;
POP;
```

CHECKSAT

The command CHECKSAT F behaves identically to QUERY $\neg F$.

RESTART

The command RESTART F can only be invoked after an invalid query. For example:

```
QUERY <formula>;
Invalid.
RESTART <formula2>;
```

The behavior of the above command will be identical to the following:

```
PUSH;
QUERY <formula>;
POP;
ASSERT <formula2>;
QUERY <formula>;
```

The advantage of using the RESTART command is that it may be much more efficient than the above command sequence. This is because when the RESTART command is used, **CVC3** will re-use what it has learned rather than starting over from scratch.

Instantiation Patterns

CVC3 processes each universally quantified formula in the current context by generating *instances* of the formula obtained by replacing its universal variables with ground terms. Patterns restrict the choice of ground terms for the quantified variables, with the goal of controlling the potential explosion of ground instances. In essence, adding patterns to a formula is a way for the user to tell **CVC3** to focus only on certain instances which, in the user's opinion, will be most helpful during a proof.

In more detail, patterns have the following effect on formulas that are found in the logical context or get later added to it while **CVC3** is trying to prove the validity of some formula φ.

If a formula in the current context starts with an existential quantifier, **CVC3** Skolemizes it, that is, replaces it in the context by the formula obtained by substituting the existentially quantified variables by fresh constants and dropping the quantifier. Any patterns for the existential quantifier are simply ignored.

If a formula starts with a universal quantifier $\mathrm{FORALL}\ (x_1 : T_1, \ldots, x_n : T_n)$, **CVC3** adds to the context a number of instances of the formula---with the goal of using them to prove the query φ valid. An instance is obtained by replacing each x_i with a ground term of the same type occurring in one of the formulas in the context, and dropping the universal quantifier. If x_i occurs in a pattern $\mathrm{PATTERN}\ (t_1, \ldots, t_m)$ for the quantifier, it will be instantiated only with terms obtained by simultaneously matching all the terms in the pattern against ground terms in the current context Γ.

Specifically, the matching process produces one or more substitutions σ for the variables in $(t_1, \ldots, t_m)$ which satisfy the following invariant: for each $i = 1, \ldots, m$, $\sigma(t_i)$ is a ground term and there is a ground term s_i in Γ such that $\Gamma \models_T \sigma(t_i) = s_i$. The variables of $(x_1 : T_1, \ldots, x_n : T_n)$ that occur in the pattern are instantiated only with those substitutions (while any remaining variables are instantiated arbitrarily).

The Skolemized version or the added instances of a context formula may themselves start with a quantifier. The same instantiation process is applied to them too, recursively.

Note that the matching mechanism is not limited to syntactic matching but is modulo the equations asserted in the context. Because of decidability and/or efficiency limitations, the matching process is not exhaustive. **CVC3** will typically miss some substitutions that satisfy the invariant above. As a consequence, it might fail to prove the validity of the query formula φ, which makes **CVC3** incomplete for contexts containing quantified formulas. It should be noted though that exhaustive matching, which can be achieved simply by not specifying any patterns, does not yield completeness anyway since the instantiation of universal variables is still restricted to just the ground terms in the context (whereas in general additional ground terms might be needed).

Subtypes

CVC3's language includes the definition of *subtypes* of value types by means of predicate subtyping.

A subtype T_p of a (sub)type T is defined as a subset of T that satisfies an associated predicate p. More precisely, if p is a term of type $T \to \mathrm{BOOLEAN}$, then for every model of p (among the models of CVC3's built-in theories), T_P is the *extension* of p, that is, the set of all and only the elements of T that satisfy the predicate p.

Subtypes like T_p above can be defined by the user with a declaration of the form:

$$subtype\_name : \mathrm{TYPE} = \mathrm{SUBTYPE}(p)$$

where p is either just a (previously declared) predicate symbol of type $T \to \mathrm{BOOLEAN}$ or a lambda abstraction of the form $\lambda x : T.\ \varphi$ where φ is any **CVC3** formula whose set of free variables contains at most x.

Here are some examples of subtype declarations:

```
Animal: TYPE;

fish : Animal;

is_fish: Animal -> BOOLEAN;

ASSERT is_fish(fish);

% Fish is a subtype of Animal:

Fish: TYPE = SUBTYPE(is_fish);

shark : Fish;

is_shark: Fish -> BOOLEAN;

ASSERT is_shark(shark);

% Shark is a subtype of Fish:

Shark: TYPE = SUBTYPE(is_shark);


% Subtypes of REAL

AllReals:     TYPE = SUBTYPE(LAMBDA (x:REAL): TRUE);

NonNegReal:   TYPE = SUBTYPE(LAMBDA (x:REAL): x >= 0);

% Subtypes of INT

DivisibleBy3: TYPE = SUBTYPE(LAMBDA (x:INT): EXISTS (y:INT): x = 3 * y);
```

CVC3 provides integers as a built-in subtype INT of $REAL$. INT is a subtype and not a base type in order to allow mixed real/integer terms without having to use coercion functions such as $\text{int\_to\_real} : \text{INT} \to \text{REAL}$ $\text{real\_to\_int} : \text{REAL} \to \text{INT}$ between terms of the two types. It is *built-in* because it is not definable by means of a first-order predicate.

Note that, with the syntax introduced so far, it seems that it may be possible to define empty subtypes, that is, subtypes with no values at all. For example:

```
NoReals:      TYPE = SUBTYPE(LAMBDA (x:REAL): FALSE);
```

However, any attempt to do this results in an error. This is because CVC3's logic assumes types are not empty. In fact, each time a subtype S is declared **CVC3** tries to prove that the subtype is non-empty; more precisely, that it is non-empty in every model of the current context. This is done simply by attempting to prove the validity of a formula of the form $\exists x : T.\, p(x)$ where T is the value type of which S is a subtype, and p is the predicate defining S. If **CVC3** succeeds, the declaration is accepted. If it fails, **CVC3** will issue a type exception and reject the declaration.

CVC3 might fail to prove the non-emptyness of a subtype either because the type is indeed empty in some models or because of CVC3's incompleteness over quantified formulas. Consider the following examples:

```
Animal: TYPE;

is_fish: Animal -> BOOLEAN;

% Fish is a subtype of Animal:

Fish: TYPE = SUBTYPE(is_fish);

Interval_0_1: TYPE = SUBTYPE(LAMBDA (x:REAL): 0 < x AND x < 1);

% Subtypes of [REAL, REAL]

StraightLine: TYPE = SUBTYPE(LAMBDA (x:[REAL,REAL]): 3*x.0 + 2*x.1 + 6 = 0);


%  Constant ARRAY subtype

ConstArray: TYPE = SUBTYPE(LAMBDA (a: ARRAY INT OF REAL):
                             EXISTS (x:REAL): FORALL (i:INT): a[i] = x);
```

Each of these subtype declarations is rejected. For instance, the declaration of `Fish` is rejected because there are models of CVC3's background theory in which `is_fish` has an empty extension. To fix that it is enough to introduce a free constant of type `Animal` and assert that it is a `Fish` as we did above.

In the case of `Interval_0_1` and `Straightline`, however, the type is indeed non-empty in every model, but **CVC3** is unable to prove it. In such cases, the user can help **CVC3** by explicitly providing a witness value for the subtype. This is done with this alternative syntax for subtype declarations:

$$\mathit{subtype\_name} : \text{TYPE} = \text{SUBTYPE}(p, t)$$

where p is again a unary predicate and t is a term (denoting an element) that satisfies p.

The following subtype declarations with witnesses are accepted by **CVC3**.

```
% Subtypes of REAL with witness

Interval_0_1: TYPE = SUBTYPE(LAMBDA (x:REAL): 0 < x AND x < 1, 1/2);

StraightLine: TYPE = SUBTYPE(LAMBDA (x:[REAL,REAL]): 3*x.0 + 2*x.1 + 6 = 0, (0, -3));
```

We observe that the declaration of ConstArray in the first example is rightly rejected under the empty context because the subtype can be empty in some models. However, even under contexts that exclude this possibility **CVC3** is still unable to to prove the subtype's non-emptyness. Again, a declaration with witness helps in this case. Example:

```
zero_array: ARRAY INT OF REAL;

ASSERT FORALL (i:INT): zero_array[i] = 0;

% At this point the context includes the constant array zero_array
% and the declaration below is accepted.

ConstArray: TYPE = SUBTYPE(LAMBDA (a: ARRAY INT OF REAL):
                             EXISTS (x:REAL): FORALL (i:INT): a[i] = x, zero_array);
```

Adding witnesses to declarations to overcome CVC3's incompleteness is an adequate, practical solution in most cases.

For additional convenience (when defining array types, for example) **CVC3** has a special syntax for specifying subtypes that are finite ranges of INT. This is however just syntactic sugar.

```
% subrange type

FiniteRangeArray: TYPE = ARRAY [-10..10] OF REAL;


% equivalent but less readable formulations

FiniteRange: TYPE = SUBTYPE(LAMBDA (x:INT): -10 <= x AND x <= 10);

FiniteRangeArray2: TYPE = ARRAY FiniteRange OF REAL;

FiniteRangeArray3: TYPE = ARRAY SUBTYPE(LAMBDA (x:INT): -10 <= x AND x <= 10) OF REAL;
```

Subtype Checking

The subtype relation between a subtype and its immediate supertype is transitive. This implies that every subtype is a subtype of some value type, and so every term can be given a unique value type. This is important because as far as type checking is concerned, subtypes are ignored by **CVC3**. By default, static type checking is enforced only at the level of maximal supertypes, and subtypes play a role only during validity checking.

In essence, for every ground term of the form $f(t_1,\ldots,t_n)$ with $i \geq 0$ in the logical context, whenever f has type $(S_1,\ldots,S_n) \rightarrow S$ where S is a subtype defined by a predicate p, **CVC3** adds to the context the assertion $p(f(t_1,\ldots,t_n))$ constraining $f(t_1,\ldots,t_n)$ to be a value in S.

This leads to correct answers by **CVC3**, provided that all ground terms are *well-subtyped* in the logical context of the query; that is, if for all terms like $f(t_1,\ldots,t_n)$ above the logical context entails that t_i is a value of S_i. When that is not the case, **CVC3** may return spurious countermodels to a query, that is, countermodels that do not respect the subtyping constraints.

For example, after the following declarations:

```
Pos: TYPE = SUBTYPE(LAMBDA (x: REAL): x > 0, 1);
Neg: TYPE = SUBTYPE(LAMBDA (x: REAL): x < 0, -1);

a: Pos;
b: REAL;

f: Pos -> Neg = LAMBDA (x:Pos): -x;
```

CVC3 will reply "Valid", as it should, to the command:

```
QUERY f(a) < 0;
```

However it will reply "Invalid" to the command:

```
QUERY f(b) < 0;
```

or to:

```
QUERY f(-4) < 0;
```

for that matter, instead of complaining in either case that the query is not well-subtyped. (The query is ill-subtyped in the first case because there are models of the empty context in which the constant b is a non-positive rational; in the second case because in all models of the context the term -4 is non-positive.)

In contrast, the command sequence

```
ASSERT b > 2*a + 3;
QUERY f(b) < 0;
```

say, produces the correct expected answer because in this case b is indeed positive in every model of the logical context.

Semantically, CVC3's behavior is justified as follows. Consider, just for simplicity (the general case is analogous), a function symbol f of type $S_1 \to T_2$ where S_1 is a subtype of some value type T_1. Instead of interpreting S_1 as *partial* function that is total over S_1 and *undefined* outside S_1, CVC3's interprets it as a *total* function from T_1 to T_2 whose behavior outside S_1 is specified in an arbitrary, but fixed, way. The specification of the behavior outside S_1 is internal to **CVC3** and can, from case to case, go from being completely empty, which means that **CVC3** will allow any possible way to extend f from S_1 to T_1, to strong enough to allow only one way to extend f. The choice depends just on internal implementation considerations, with the understanding that the user is not really interested in f's behavior outside S_1 anyway.

A simple example of this approach is given by the arithmetic division operation /. Mathematically division is a partial function from $\mathrm{REAL} \times \mathrm{REAL}$ to REAL undefined over pairs in $\mathrm{REAL} \times \{0\}$. **CVC3** views / as a total function from $\mathrm{REAL} \times \mathrm{REAL}$ to REAL that maps pairs in $\mathrm{REAL} \times \{0\}$ to 0 and is defined as usual otherwise. In other words, **CVC3** extends the theory of rational numbers with the axiom $\forall\, x : \mathrm{REAL}.\ x/0 = 0$. Under this view, queries like

```
x: REAL;

QUERY x/0 = 0 ;
QUERY 3/x = 3/x ;
```

are perfectly legitimate. Indeed the first formula is valid because in each model of the empty context, x/0 is interpreted as zero and = is interpreted as the identity relation. The second formula is valid, more generally, because for each interpretation of x the two arguments of = will evaluate to the same rational number. **CVC3** will answer accordingly in both cases.

While this behavior is logically correct, it may be counter-intuitive to users, especially in applications that intend to give **CVC3** only well-subtyped formulas. For these applications it is more useful to the user to get a type error from **CVC3** as soon as it receives an ill-subtyped assertion or query, such as for instance the two queries above. This feature is provided in **CVC3** by using the command-line option +tcc. The mechanism for checking well-subtypedness is described below.

Type Correctness Conditions

CVC3 uses an algorithm based on Type Correctness Conditions, TCCs for short, to determine if a term or formula is well-subtyped. This of course requires first an adequate notion of well-subtypedness. To introduce that

notion, let us start with the following definition where T is the union of CVC3's background theories.

Let us say that a (well-typed) term t containing no proper subterms of type $\mathrm{BOOLEAN}$ is *well-subtyped in a model* M of T (assigning an interpretation to all the free symbols and free variables of t) if

- t is a constant or a variable, or
- it is of the form $f(t_1, \ldots, t_n)$ where f has type $(S_1, \ldots, S_n) \to S$ and each t_i is well-subtyped in M and interpreted as a value of S_i.

Note that this inductive definition includes the case in which the term is an atomic formula. Then we can say that an atomic formula is well-subtyped in a logical context Γ if it is well-subtyped in every model of Γ and T.

While this seems like a sensible definition of well-subtypedness for atomic formulas, it is not obvious how to extend it properly to non-atomic formulas. For example, defining a non-atomic formula to be well-subtyped in a model if all of its atoms are well-subtyped is too stringent. Perfectly reasonable formulas like

$$y > 0 \Rightarrow x/y = z$$

with x, y, and z free constants (or free variables) of type REAL, say, would not be well-subtyped in the empty context because there are models of T in which the atom $x/y = z$ is not well-subtyped (namely, those that interpret y as zero).

A better definition can be given by treating logical connectives *non-strictly* with respect to ill-subtypedness. More formally, but considering for simplicity only formulas built with atoms, negation and disjunction connectives, and existential quantifiers (the missing cases are analogous), we define a non-atomic formula ϕ to be well-subtyped in a model M of T if one of the following holds:

- ϕ has the form $\neg\phi_1$ and ϕ_1 is well-subtyped in M;
- ϕ has the form $\phi_1 \vee \phi_2$ and (i) both ϕ_1 and ϕ_2 are well-subtyped in M or (ii) ϕ_1 holds and is well-subtyped in M or (iii) ϕ_2 holds and is well-subtyped in M;
- ϕ has the form $\exists x.\, \phi_1$ and (i) ϕ_1 holds and is well-subtyped in some model M' that differs from M at most in the interpretation of x or (ii) ϕ_1 is well-subtyped in every such model M'.

In essence, this definition is saying that for well-subtypedness in a model it is irrelevant if a formula ϕ has an ill-subtyped subformula, as long as the truth value of ϕ is independent from the truth value of that subformula.

Now we can say in general that a **CVC3** formula is *well-subtyped in a context* Γ if it is well-subtyped in every model of Γ and T.

According to this definition, the previous formula $y > 0 \Rightarrow x/y = z$, which is equivalent to $\neg(y > 0) \vee x/y = z$, is well-subtyped in the empty context. In fact, in all the models of T that interpret y as zero, the subformula $\neg(y > 0)$ is true and well-subtyped; in all the others, both $\neg(y > 0)$ and $x/y = z$ are well-subtyped.

This notion of well-subtypedness has a number of properties that make it fairly robust. One is that it is invariant with respect to equivalence in a context: for every context Γ and formulas ϕ, ϕ' such that $\Gamma \models_T \phi \Leftrightarrow \phi'$, the first formula is well-subtyped in Γ if and only if the second is.

Perhaps the most important property, however, is that the definition can be effectively reflected into CVC3's logic itself: there is a procedure that for any **CVC3** formula ϕ can compute a well-subtyped formula Δ_ϕ, a *type correctness condition* for ϕ, such that ϕ is well-subtyped in a context Γ if and only if $\Gamma \models_T \Delta_\phi$. This has the nice consequence that the very inference engine of **CVC3** can be used to check the well-subtypedness of **CVC3** formulas.

When called with the TCC option on (by using the command-line option `+tcc`), **CVC3** behaves as follows. Whenever it receives an ASSERT or QUERY command, the system computes the TCC of the asserted formula or query and checks its validity in the current context (for ASSERTs, before the formula is added to the logical context). If it is able to prove the TCC valid, it just adds the asserted formula to the context or checks the validity of the query formula. If it is unable to prove the TCC valid, it raises an ill-subtypedness exception and aborts.

It is worth pointing out that, since **CVC3** checks the validity of an asserted formula in the current logical context at the time of the assertion, the order in which formulas are asserted makes a difference. For instance,

attempting to enter the following sequence of commands:

```
f: [0..100] -> INT;
x: [5..10];
y: REAL;

ASSERT f(y + 3/2) < 15;
ASSERT y + 1/2 = x;
```

results in a TCC failure for the first assertion because the context right before it does not entail that the term `y + 3/2` is in the range `0..100`. In contrast, the sequence

```
f: [0..100] -> INT;
x: [5..10];
y: REAL;

ASSERT y + 1/2 = x;
ASSERT f(y + 3/2) < 15;
```

is accepted because each of the formulas above is well-subtyped at the time of its assertion. Note that the assertion of both formulas together in the empty context with

```
ASSERT f(y + 3/2) < 15 AND y + 1/2 = x
```

or with

```
ASSERT y + 1/2 = x AND f(y + 3/2) < 15
```

is also accepted because the conjunction of the two formulas is well-subtyped in the empty context.

SMT-LIB Input Language

CVC3 is able to read and execute queries in the SMT-LIB format.

Specifically, when called with the option `-lang smt` it accepts as input an SMT-LIB *benchmark* belonging to one of the SMT-LIB *sublogics*. For a well-formed input benchmark, **CVC3** returns the string "sat", "unsat" or "unknown", depending on whether it can prove the benchmark satisfiable, unsatisfiable, or neither.

At the time of this writing **CVC3** supported all SMT-LIB sublogics.

We refer the reader to the SMT-LIB website for information on SMT-LIB, its formats, its logics, and its on-line library of benchmarks.